Programming Languages: Design and Implementation

Functional Programming

CE 40364

Fall 1399

Session 3

PROGRAM M ING PARADIGM S

Programming Paradigms

- During the next few weeks we are going to work with functional programming. Before I can explain to you what FP is, I thought I'd better put things into perspective by talking about other programming paradigms.
- Over the last 40 or so years, a number of programming paradigms (a programming paradigm is a way to think about programs and programming) have emerged.

Programming Paradigms. . .

A programming paradigm

- is a way to think about programs, programming, and problem solving,
- is supported by one or more programming languages.

Being familiar with several paradigms makes you a better programmer and problem solver. The most popular paradigms:

1. Imperative programming.
2. Functional programming.
3. Object-oriented programming.
4. Logic Programming.

When all you have is a hammer, everything looks like a nail.

Programming Paradigms. . .

Imperative Programming

- Programming with state.
- Also known as procedural programming. The first to emerge in the 1940s-50s. Still the way most people learn how to program.
- FORTRAN, Pascal, C, BASIC.

Functional Programming

- Programming with values.
- Arrived in the late 50s with the LISP language. LISP is still popular and widely used by AI people.
- LISP, Miranda, Haskell, Gofer.

Programming Paradigms. . .

Object-Oriented Programming

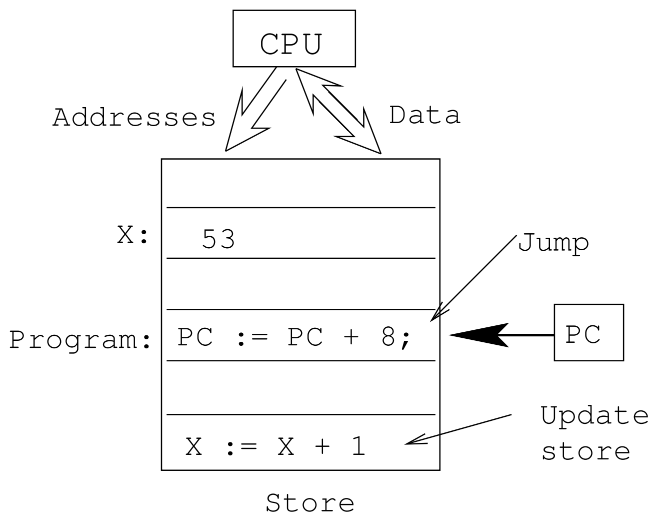
- Programming with objects that encapsulate data and operations.
- A variant of imperative programming first introduced with the Norwegian language Simula in the mid 60s.
- Simula, Eiffel, Modula-3, C++.

Logic Programming

- Programming with relations.
- Introduced in the early 70s. Based on predicate calculus. Prolog is popular with Computational Linguists.
- Prolog, Parlog.

Procedural Programming

We program an abstraction of the Von Neumann Machine, consisting of a store (memory), a program (kept in the store), A CPU and a program counter (PC):



Computing `x:=x+1`

1. Compute `x`'s address, send it to the store, get `x`'s value back.
2. Add 1 to `x`'s value.
3. Send `x`'s address and new value to the store for storage.
4. Increment `PC`.

Procedural Programming...

The programmer...

- uses control structures (IF, WHILE, ...) to alter the program counter (PC),
- uses assignment statements to alter the store.
- is in charge of memory management, i.e. declaring variables to hold values during the computation.

```
function fact (n:integer):integer;
var s,i : integer := 1;
begin
   while i<=n do s:=s*i; i:=i+1; end;
   return s;
end fact.
```

Procedural Programming…

Procedural programming is difficult because:

1. A procedural program can be in a large number of states. (Any combination of variable values and PC locations constitutes a possible state.) The programmer has to keep track of all of them.
2. Any global variable can be changed from any location in the program. (This is particularly true of languages like C & C++ [Why?]).
3. It is difficult to reason mathematically about a procedural program.

Functional Programming

In contrast to procedural languages, functional programs don’t concern themselves with state and memory locations. Instead, they work exclusively with values, and expressions and functions which compute values.

- Functional programming is not tied to the von Neumann machine.
- It is not necessary to know anything about the underlying hardware when writing a functional program, the way you do when writing an imperative program.
- Functional programs are more declarative than procedural ones; i.e. they describe what is to be computed rather than how it should be computed.

Functional Languages

Common characteristics of functional programming languages:

1. Simple and concise syntax and semantics.
2. Repetition is expressed as recursion rather than iteration.
3. Functions are first class objects. I.e. functions can be manipulated just as easily as integers, floats, etc. in other languages.
4. Data as functions. I.e. we can build a function on the fly and then execute it. (Some languages).

Functional Languages. . .

5. Higher-order functions. I.e. functions can take functions as arguments and return functions as results.
6. Lazy evaluation. Expressions are evaluated only when needed. This allows us to build infinite data structures, where only the parts we need are actually constructed. (Some languages).
7. Garbage Collection. Dynamic memory that is no longer needed is automatically reclaimed by the system. GC is also available in some imperative languages (Modula-3, Eiffel) but not in others (C, C++, Pascal).

Functional Languages. . .

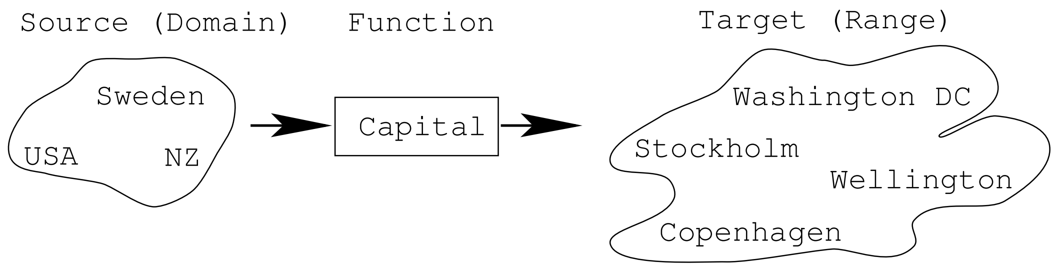
8. Polymorphic types. Functions can work on data of different types. (Some languages).
9. Functional programs can be more easily manipulated mathematically than procedural programs.

<ins>Pure vs. Impure FPL</ins>

- Some functional languages are pure, i.e. they contain no imperative features at all. Examples: Haskell, Miranda, Gofer.
- Impure languages may have assignment-statements, goto:s, while-loops, etc. Examples: LISP, ML, Scheme.

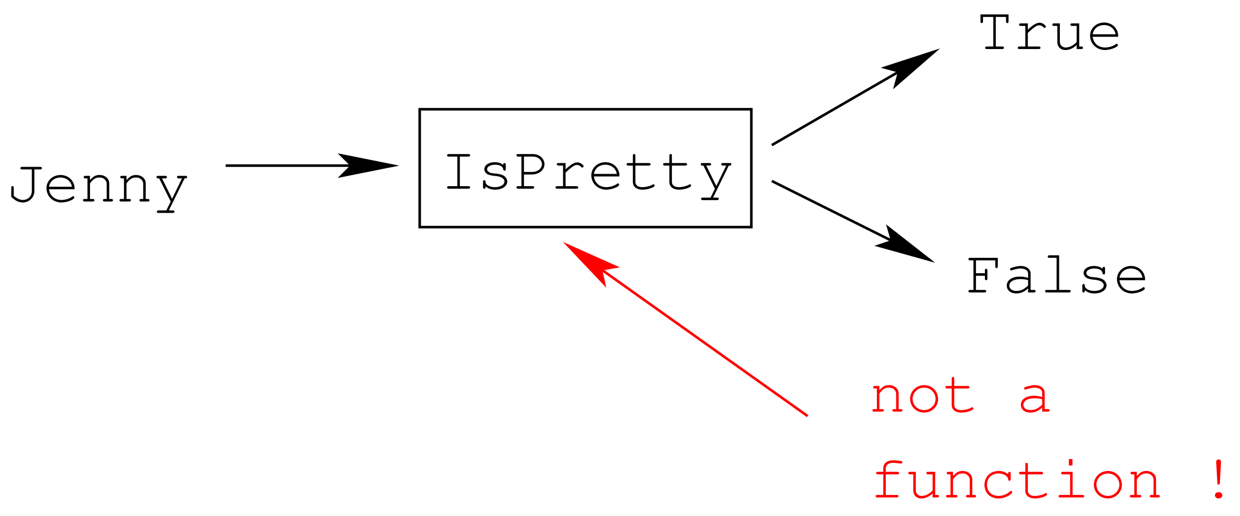
What is a function?

- A function maps argument values (inputs) to result values (outputs).
- A function takes argument values from a source set (or domain).
- A function produces result values that lie in a target set (or range).



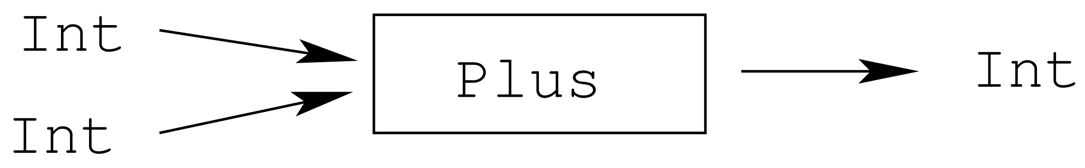
More on functions

- A function must not map an input value to more than one output value. Example:



More on functions. . .

- If a function F maps every element in the domain to some element in the range, then F is total. I.e. a total function is defined for all arguments.



More on functions…

- A function that is undefined for some inputs, is called partial.

$$\begin{tikzcd}
\texttt{Int} \arrow[dr] & & \\
& \boxed{\texttt{Divide}} \arrow[r] & \texttt{Int} \\
\texttt{Int} \arrow[ur] & &
\end{tikzcd}$$

- `Divide` is partial since $\frac{?}{0} = ?$ is undefined.

Specifying functions

A function can be specified extensionally or intentionally.

Extensionally:

- Enumerate the elements of the (often infinite) set of pairs "`(argument, result)`" or "`Argument` ↦ `Result`."
- The extensional view emphasizes the external behavior (or specification), i.e. what the function does, rather than how it does it.

```
double = {⋯, (1,2), (5,10), ⋯}
even = {⋯, (0,True), (1,False), ⋯}
double = {⋯, 1↦2, 5↦10, ⋯}
isHandsome={Chris↦True,Hugh↦False}
```

Specifying functions…

Intensionally:

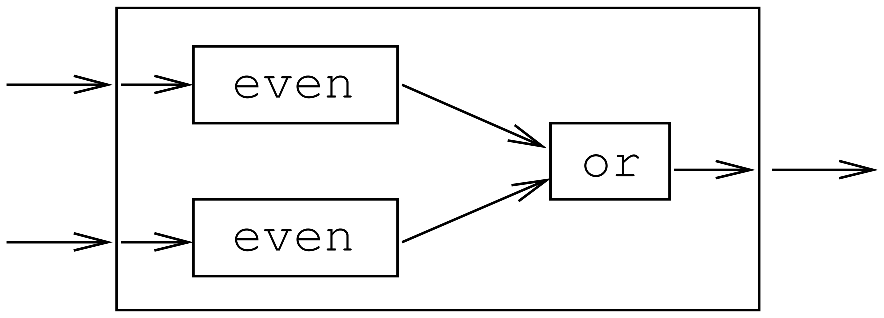
- Give a rule (i.e. algorithm) that computes the result from the arguments.
- The intentional view emphasizes the process (or algorithm) that is used to compute the result from the arguments.

```
double x = 2 * x
even x = x mod 2 == 0
isHandsome x = if isBald x
         then True
         else False
```

Specifying functions. . .

Graphically:

- The graphical view is a notational variant of the intentional view.



eithereven

Function Application

- The most important operation in a functional program is function application, i.e. applying an input argument to the function, and retrieving the result:

```
double x = 2 * x
even x = x mod 2 == 0

double 5 ⇒ 10
even 6 ⇒ True
```

Function Composition

- Function composition makes the result of one function application the input to another application:

```
double x = 2 * x
even x = x mod 2 == 0

even (double 5) ⇒ even 10 ⇒ True
```

Function Definition — Example

Example: How many numbers are there between m and n, nclusive?

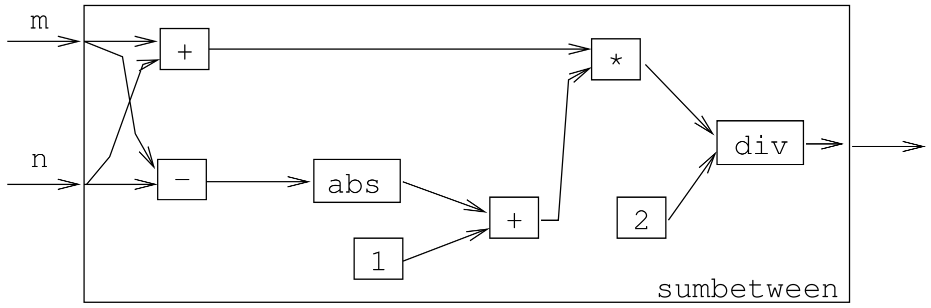
Extensional Definition:

sumbetween m n = $\{\cdots\ (1,1) \mapsto 1, (1,2) \mapsto 2, \cdots, (2,10) \mapsto 9\}$

Intentional Definition:

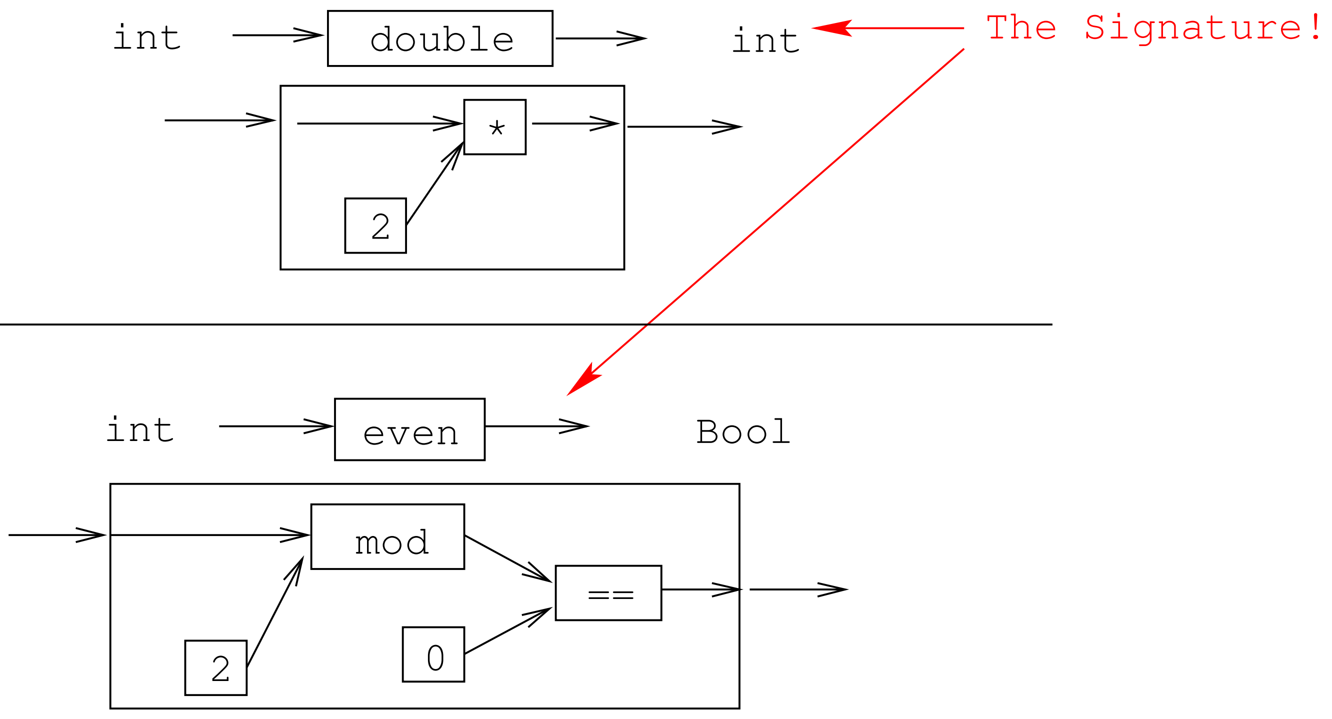
```
sumbetween m n = ((m + n) * (abs (m-n) + 1)) div 2
```

Graphical Definition:



Function Signatures

To define a function we must specify the types of the input and output sets (domain and range, i.e. the function's signature), and an algorithm that maps inputs to outputs.



Referential Transparency

- The most important concept of functional programming is referential transparency. Consider the expression

$$(2 * 3) + 5 * (2 * 3)$$

- $(2 * 3)$ occurs twice in the expression, but it has the same meaning (6) both times.
- RT means that the value of a particular expression (or sub-expression) is always the same, regardless of where it occurs.
- This concept occurs naturally in mathematics, but is broken by imperative programming languages.
- RT makes functional programs easier to reason about mathematically.

Referential Transparency...

- RT is a fancy term for a very simple concept.
- What makes FP particularly simple, direct, and expressive, is that there is only one mechanism (function application) for communicating between subprograms.
- In imperative programs we can communicate either through procedure arguments or updates of global variables. This is hard to reason about mathematically.
- A notation (programming language) where the value of an expression depends only on the values of the sub-expressions is called referentially transparent.

Referential Transparency...

- Pure functional programming languages are referentially transparent.
- This means that it is easy to find the meaning (value) of an expression.
- We can evaluate it by substitution. I.e. we can replace a function application by the function definition itself.

Referential Transparency...

Evaluate $\boxed{\texttt{even (double 5)}}$:

```
double x = 2 * x
even x = x mod 2 == 0
```

```
even (double 5) ⇒
        even (2 * 5) ⇒
        even 10 ⇒
        10 mod 2 == 0 ⇒
        0 == 0 ⇒ True
```

Referential Transparency…

So, isn't Pascal referentially transparent??? Well, sometimes, yes, but not always. If a Pascal function f has side-effects (updating a global variable, doing input or output), then $f(3) + f(3)$ may not be the same as $2 * f(3)$. I.e. The second $f(3)$ has a different meaning than the first one.

```
var G : integer;
function f (n:integer) :  integer;
begin G:=G+1; f:=G+n; end;
```

```
begin
   G := 0;
   print f(3)+f(3);
   {prints 4+5=9}
end.
```

```
begin
   G := 0;
   print 2*f(3);
   {prints 2*4=8}
end.
```

Referential Transparency. . .

Furthermore, in many imperative languages the order in which the arguments to a binary operator are evaluated are undefined.

```
var G : integer;
function f (n:integer) :  integer;
begin G:=G+1; f:=G+n; end;
```

Left f(3) evaluated first.

```
begin
   G := 0;
   print f(3)-f(4);
   prints 4-6=-2
end.
```

Right f(3) evaluated first.

```
begin
   G := 0;
   print f(3)-f(4);
   prints 5-5=0
end.
```

Referential Transparency…

This cannot happen in a pure functional language.

1. Expressions and sub-expressions always have the same value, regardless of the environment in which they're evaluated.
2. The order in which sub-expressions are evaluated doesn't effect the final result.
3. Functions have no side-effects.
4. There are no global variables.

Referential Transparency…

5. Variables are similar to variables in mathematics: they hold a value, but they can't be updated.
6. Variables aren't (updatable) containers the way they are imperative languages.
7. Hence, functional languages are much more like mathematics than imperative languages. Functional programs can be treated as mathematical text, and manipulated using common algebraic laws.

Homework

- Here is a mathematical definition of the combinatorial function $\binom{n}{r}$ "n choose r", which computes the number of ways to pick r objects from n:

$$\left(\begin{array}{c} n \\ r \end{array}\right) = \frac{n!}{r! * (n-r)!}$$

- Give an extensional, intentional, and graphical definition of the combinatorial function, using the notations suggested in this lecture.
- You may want to start by defining an auxiliary function to compute the factorial function, $n! = 1 * 2 * \cdots * n$.

Scheme & Racket - Introduction

Background

- Scheme is based on LISP which was developed by John McCarthy in the mid 50s.
- LISP stands for *LISt Processing*, not *Lots of Irritating Silly Parentheses*.
- Functions and data share the same representation: S-Expressions.
- A basic LISP implementation needs six functions `cons, car, cdr, equal, atom, cond`.
- Scheme was developed by Sussman and Steele in 1975.

S-Expressions

- An S-Expression is a balanced list of parentheses.

More formally, an S-expression is

1. a literal (i.e., number, boolean, symbol, character, string, or empty list).
2. a list of s-expressions.

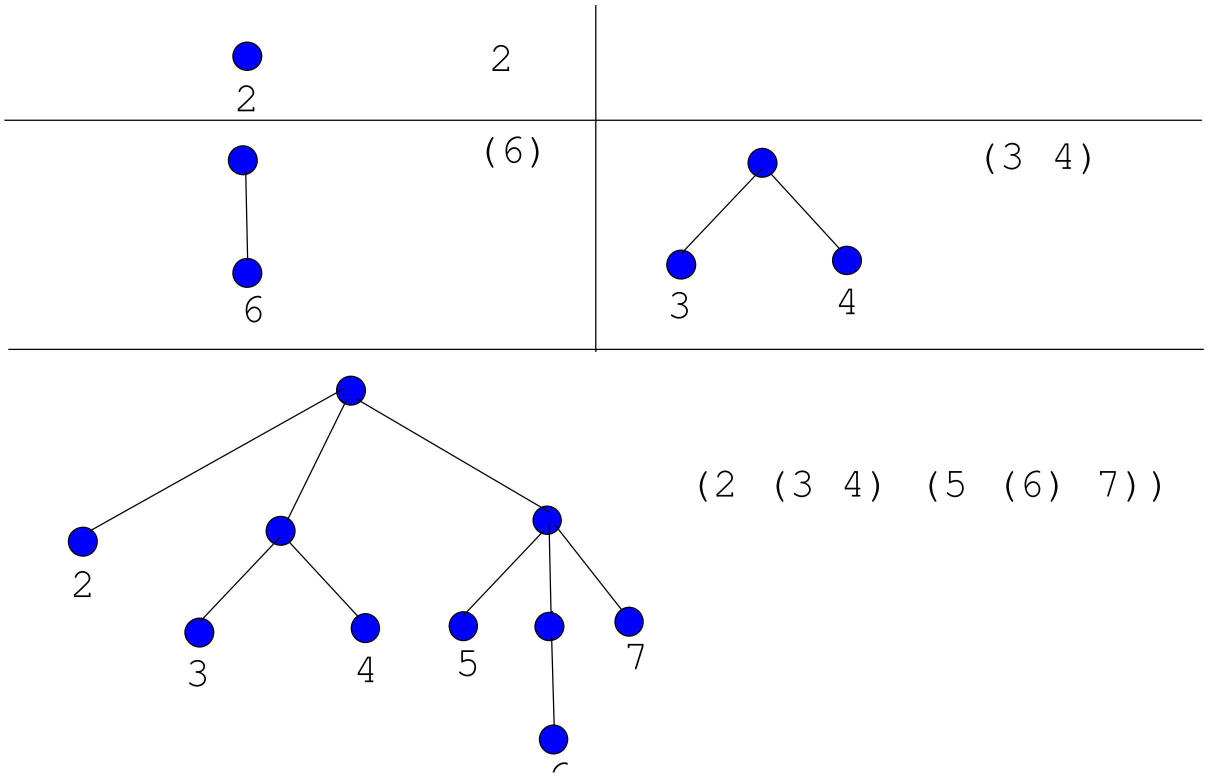
- Literals are sometimes called atoms.

S-Expressions — Examples

Legal	Illegal
`66`	`(`
`()`	`(5))`
`(4 5)`	`()()`
`((5))`	`(4 (5)`
`(()())`	`)(`
`((4 5) (6 (7)))`	

S-Expressions as Trees

- An S-expression can be seen as a linear representation of tree-structure:



S-Expressions as Function Calls

- A special case of an S-expression is when the first element of a list is a function name.
- Such an expression can be evaluated.

```
> (+ 4 5)
9
> (add-five-to-my-argument 20)
25
> (draw-a-circle 20 45)
#t
```

S-Expressions as Functions

- As we will see, function definitions are also S-expressions:

```
(define (farenheit-2-celsius f)
  (* (- f 32) 5/9)
)
```

- So, Scheme really only has one syntactic structure, the S-expression, and that is used as a data-structure (to represent lists, trees, etc), as function definitions, and as function calls.

Function Application

- In general, a function application is written like this:

$$(\texttt{operator}\ \texttt{arg}_1\ \texttt{arg}_2\ \ldots\ \texttt{arg}_n)$$

- The evaluation proceeds as follows:
  1. Evaluate `operator`. The result should be a function $\mathcal{F}$.
  2. Evaluate

     $$\texttt{arg}_1,\ \texttt{arg}_2,\ \ldots\ \texttt{arg}_n$$

     to get

     $$\texttt{val}_1,\ \texttt{val}_2,\ \ldots\ \texttt{val}_n$$

  3. Apply $\mathcal{F}$ to $\texttt{val}_1,\ \texttt{val}_2,\ \ldots\ \texttt{val}_n$.

Function Application — Examples

```
> (+ 4 5)
9
> (+ (+ 5 6) 3)
14
> 7
7
> (4 5 6)
eval:  4 is not a function
> #t
#t
```

Atoms — Numbers

Scheme has

- Fractions (`5/9`)
- Integers (`5435`)
- Complex numbers (`5+2i`)
- Inexact reals (`#i3.14159265`)

```
> (+ 5 4)
9
> (+ (* 5 4) 3)
23
> (+ 5/9 4/6)
1.2
> 5/9
0.5
```

Atoms — Numbers. . .

```
> (+ 5/9 8/18)
1
> 5+2i
5+2i
> (+ 5+2i 3-i)
8+1i
> (* 236542164521634 3746573426573425643)
886222587860913289285513763860662
> pi
#i3.141592653589793
> e
#i2.718281828459045
> (* 2 pi)
#i6.283185307179586
```

Atoms — Numbers. . .

- Scheme tries to do arithmetic exactly, as much as possible.
- Any computations that depend on an inexact value becomes inexact.
- Scheme has many builtin mathematical functions:

```
> (sqrt 16)
4
> (sqrt 2)
#i1.4142135623730951
> (sin 45)
#i0.8509035245341184
> (sin (/ pi 2))
#i1.0
```

Atoms — Strings

- A string is enclosed in double quotes.

```
> (display "hello")
hello
> "hello"
"hello"
> (string-length "hello")
5
> (string-append "hello" " " "world!")
"hello world!"
```

Atoms — Booleans

- `true` is written `#t`.
- `false` is written `#f`.

```
> #t
true
> #f
false
> (display #t)
#t
> (not #t)
false
```

Identifiers

- Unlike languages like C and Java, Scheme allows identifiers to contain special characters, such as `! $ % & * + - . / : < = > ? @ ^ _ ~`. Identifiers should not begin with a character that can begin a number.
- This is a consequence of Scheme's simple syntax.
- You couldn't do this in Java because then there would be many ways to interpret the expression `X-5+Y`.

Legal	Illegal
`h-e-l-l-o`	`3some`
`give-me!`	`-stance`
`WTF?`	

Defining Variables

- define binds an expression to a global name:

```
(define name expression)
```

```
(define PI 3.14)

> PI
3.14

(define High-School-PI (/ 22 7))

> High-School-PI
3.142857
```

Defining Functions

- define binds an expression to a global name:

  (define (name arg_1 arg_2 ...) *expression*)

- arg_1 arg_2 ... are formal function parameters.

```
(define (f) 'hello)

> (f)
hello

(define (square x) (* x x))

> (square 3)
9
```

Defining Helper Functions

- A Scheme program consists of a large number of functions.
- A function typically is defined by calling other functions, so called helper or auxiliary functions.

```
(define (square x) (* x x))

(define (cube x) (* x (square x)))

> (cube 3)
27
```

Preventing Evaluation

- Sometimes you don't want an expression to be evaluated.
- For example, you may want to think of (+ 4 5) as a list of three elements `+`, `4`, and `5`, rather than as the computed value `9`.
- (quote (+ 4 5)) prevents `(+ 4 5)` from being evaluated. You can also write '(+ 4 5).

```
> (display (+ 4 5))
9
> (display (quote (+ 4 5)))
(+ 4 5)
> (display '(+ 4 5))
(+ 4 5)
```

Scheme so Far

- A function is defined by

```
    (define (name arguments) expression)
```

- A variable is defined by

```
              (define name expression)
```

- Strings are inclosed in double quotes, like `"this"`. Common operations on strings are
  - (string-length `string`)
  - (string-append `list-of-strings`)
- Numbers can be exact integers, inexact reals, fractions, and complex. Integers can get arbitrarily large.
- Booleans are written `#t` and `#f`.

Scheme so Far. . .

- An inexact number is written: `#i3.14159265`.
- Common operations on numbers are
  - (+ `arg1 arg2`), (- `arg1 arg2`)
  - (add1 `arg`), (sub1 `arg`)
  - (min `arg1 arg2`), (max `arg1 arg2`)
- A function application is written:

```
> (function-name arguments)
```

- Quoting is used to prevent evaluation

```
(quote argument)
```

  or

```
'argument
```

CONDITIONAL EXPRESSIONS

Comparison Functions

- Boolean functions (by convention) end with a ?.
- We can discriminate between different kinds of numbers:

```
> (complex?  3+4i)
   #t
> (complex?  3)
   #t
> (real?  3)
   #t
> (real?  -2.5+0.0i)
   #t
> (rational?  6/10)
```

Comparison Functions. . .

```
    #t
> (rational?  6/3)
    #t
> (integer?  3+0i)
    #t
> (integer?  3.0)
    #t
> (integer?  8/4)
    #t
```

Tests on Numbers

- Several of the comparison functions can take multiple arguments.
- `(< 4 5 6 7 9 234)` returns true since the numbers are monotonically increasing.

```
> (< 4 5)
true
> (< 4 5 6 7 9 234)
true
> (> 5 2 1 3)
false
> (= 1 1 1 1 1)
true
> (<= 1 2 2 2 3)
true
```

Tests on Numbers…

```
> (>= 5 5)
true
> (zero?  5)
false
> (positive?  5)
true
> (negative?  5)
false
> (odd?  5)
true
> (even?  5)
false
```

Conditionals — If

- If the `test-expression` evaluates to `#f` (False) return the valuen of the `then-expression`, otherwise return the value of the `else-expression`:

```
(if test-expression
   then-expression
   else-expression
)
```

- Up to language level “Advanced Student” if-expressions must have two parts.
- Set the language level to Standard (R5RS) to get the standard Scheme behavior, where the else-expression is optional.

Conditionals — If…

```
> (define x 5)
> (if (= x 5) 2 4)
2
> (if (< x 3)
         (display "hello")
         (display "bye"))
bye
> (display
         (if (< x 3) "hello" "bye"))
bye
```

If it's not False (#f), it's True (#t)

- Any value that is not false, is interpreted as true.
- NOTE: In DrScheme this depends on which language level you set. Up to "Advanced Student", the `test-expression` of an `if` must be either `#t` or `#f`.
- Set the language level to Standard (R5RS) to get the standard Scheme behavior:

```
> (if 5 "hello" "bye")
"hello"
> (if #f "hello" "bye")
"bye"
> (if #f "hello")
> (if #t "hello")
"hello"
```

Boolean Operators

- `and` and `or` can take multiple arguments.
- `and` returns true if none of its arguments evaluate to False.
- `or` returns true if any of its arguments evaluates to True.

```
> (and (< 3 5) (odd?  5) (inexact?  (cos 32)))
#t
> (or (even?  5) (zero?  (- 5 5)))
#t
> (not 5)
#f
> (not #t)
#f
```

Boolean Operators. . .

- In general, any value that is not `#f` is considered true.
- `and` and `or` evaluate their arguments from left to right, and stop as soon as they know the final result.
- The last value evaluated is the one returned.

```
> (and "hello")
"hello"
> (and "hello" "world")
"world"
> (or "hello" "world")
"hello"
```

Defining Boolean Functions

- We can define our own boolean functions:

```
(define (big-number?  n)
      (> n 10000000)
)

> (big-number?  5)
#f
> (big-number?  384783274832748327)
#t >
```

Conditionals — `cond`

- `cond` is a generalization of `if`:

```
(cond
    (cond-expression₁ result-expression₁)
    (cond-expression₂ result-expression₂)
    ...
    (else else-expression))
```

- Each `cond-expression`$_i$ is evaluated in turn, until one evaluates to not False.

```
> (cond
            ((< 2 3) 4)
            ((= 2 3) 5)
            (else 6))
4
```

Conditionals — cond...

- To make this a bit more readable, we use square brackets around the cond-clauses:

```
(cond
   [cond-expr1 result-expr1]
   [cond-expr2 result-expr2]
   ...
   [else else-expression])
```

```
> (cond [#f 5] [#t 6])
6
> (cond
    [(= 4 5) "hello"]
    [(> 4 5) "goodbye"]
    [(< 4 5) "see ya!"])
"see ya!"
```

Conditionals — `case`

- `case` is like Java/C's `switch` statment:

```
(case key
   [(expr_1 expr_2 ...)   result-expr_1]
   [(expr_11 expr_11 ...)   result-expr_2]
   ...
   (else else-expr))
```

- The *key* is evaluated once, and compared against each *cond-expr* in turn, and the corresponding *result-expr* is returned.

```
> (case 5 [(2 3) "hello"] [(4 5) "bye"])
"bye"
```

Conditionals — case...

```
(define (classify n)
   (case n
      [(2 4 8 16 32) "small power of 2"]
      [(2 3 5 7 11) "small prime number"]
      [else "some other number"]
   )
 )
> (classify 4)
"small power of 2"
> (classify 3)
"small prime number"
> (classify 2)
"small power of 2"
> (classify 32476)
"some other number"
```

Sequencing

- To do more than one thing in sequence, use `begin`:

  `(begin` arg_1 arg_2 `...)`

```
> (begin
    (display "the meaning of life=")
    (display (* 6 7))
    (newline)
  )
the meaning of life=42
```

Examples — $!n$

- Write the factorial function $!n$:

```
(define (!  n)
    (cond
        [(zero?  n) 1]
        [else (* n (!  (- n 1)))]
    )
)

> (!  5)
120
```

Examples — $\binom{n}{r}$

- Write the $\binom{n}{r}$ function in Scheme:

$$\binom{n}{r} = \frac{n!}{r! * (n-r)!}$$

- Use the factorial function from the last slide.

```
(define (choose n r)
   (/ (!  n) (* (!  r) (!  (- n r))))
)

> (choose 5 2)
10
```

Examples — `(sum m n)`

- Write a function `(sum m n)` that returns the sum of the integers between `m` and `n`, inclusive.

```
(define (sum m n)
    (cond
        [(= m n) m]
        [else (+ m (sum (+ 1 m) n))]
    )
)

> (sum 1 2)
3
> (sum 1 4)
10
```

Examples — Ackermann’s function

- Implement Ackermann’s function:

$$
\begin{aligned}
A(1, j) &= 2j \text{ for } j \geq 1 \\
A(i, 1) &= A(i-1, 2) \text{ for } i \geq 2 \\
A(i, j) &= A(i-1, A(i, j-1)) \text{ for } i, j \geq 2
\end{aligned}
$$

```
(define (A i j)
   (cond
      [(and (= i 1) (>= j 1)) (* 2 j)]
      [(and (>= i 2) (= j 1)) (A (- i 1) 2)]
      [(and (>= i 2) (>= j 2))
            (A (- i 1) (A i (- j 1)))]
   )
)
```

Examples — Ackermann's function. . .

- Ackermann's function grows very quickly:

```
> (A 1 1)
2
> (A 3 2)
512
> (A 3 3)
1561585988519419914804999641169225
4958731641184786755447122887443528
0601470939536037485963338068553800
6371637297210170750776562389313989
2867298012168192
```

Scheme so Far

- Unlike languages like Java and C which are statically typed (we describe in the program text what type each variable is) Scheme is dynamically typed. We can test at runtime what particular type of number an atom is:
  - `(complex? arg), (real? arg)`
  - `(rational? arg), (integer? arg)`
- Tests on numbers:
  - `(< arg1, arg2), (> arg1, arg2)`
  - `(= arg1, arg2), (<= arg1, arg2)`
  - `(>= arg1, arg2), (zero? arg)`
  - `(positive? arg), (negative? arg)`
  - `(odd? arg), (even? arg)`

Scheme so Far…

- Unlike many other languages like Java which are statement-oriented, Scheme is expression-oriented. That is, every construct (even `if`, `cond`, etc) return a value. The `if-expression` returns the value of the `then-expr` or the `else-expr`:

  ```
      (if test-expr then-expr else-expr)
  ```

  depending on the value of the `test-expr`.

Scheme so Far…

- The `cond`-expression evaluates its guards until one evaluates to non-false. The corresponding value is returned:

```
(cond
    (guard₁ value₁)
    (guard₂ value₂)
    ...
    (else else-expr))
```

Scheme so Far…

- The `case`-expression evaluates `key`, finds the first matching expression, and returns the corresponding result:

```
(case key
   [(expr1  expr2  ...)   result-expr1]
   [(expr11  expr11  ...)   result-expr2]
   ...
   (else else-expr))
```

Scheme so Far…

- `and` and `or` take multiple arguments, evaluate their results left-to-right until the outcome can be determined (for `or` when the first non-`false`, for `and` when the first `false` is found), and returns the last value evaluated.

SYMBOLS AND STRUCTURES

Symbols

- In addition to numbers, strings, and booleans, Scheme has a primitive data-type (*atom*) called symbol.
- A symbol is a lot like a string. It is written:

```
'identifier
```

- Here are some examples:

```
'apple
'pear
'automobile
```

- `(symbol? arg)` checks if an atom is a symbol.
- To compare two symbols for equality, use `(eq? arg1 arg2)`. `HTDP` says to use `(symbol=? arg1 arg2)` but DrScheme doesn't seem to support this.

Symbols…

```
> (symbol?  "hello")
#f
> (symbol?  'apple)
#t
> (eq?  'a 'a)
#t
> (eq?  'a 'b)
#f
> (display 'apple)
apple
> (string->symbol "apple")
apple
> (symbol->string 'apple)
"apple"
```

Symbols...

```
(define (healthy?  f)
    (case f
        [(sushi sashimi) 'hell-yeah]
        [(coke) 'I-wish]
        [(licorice) 'no-but-yummy]
        [else 'nope]
    ))
> (healthy?  'sashimi)
hell-yeah
> (healthy?  'coke)
i-wish
> (healthy?  'licorice)
no-but-yummy
> (healthy?  'pepsi)
nope
```

Structures

- Some versions of Scheme have structures. Select Advanced Student in DrScheme.
- These are similar to C's `struct`, and Java's `class` (but without inheritance and methods).
- Use `define-struct` to define a structure:

```
(define-struct struct-name (f1 f2 ...))
```

- `define-struct` will automatically define a constructor:

```
(make-struct-name (f1 f2 ...))
```

  and field-selectors:

```
struct-name-f1
struct-name-f2
```

Structures…

```
(define-struct person (name sex date-of-birth))

> (define bob (make-person "bob" 'male '1978))
> bob
(make-person "bob" 'male '1978)
> (define alice (
          make-person "alice" 'female '1979))

> (person-sex bob)
'male
> (person-date-of-birth alice)
'1979
```

Equivalence

- Every language definition has to struggle with equivalence; i.e. what does it mean for two language elements to be the same?
- In Java, consider the following example:

```
void M(String s1, String s2, int i1, int i2) {
    if (i1 == i2) ...;
    if (s1 == s2) ...;
    if (s1.equals(s2)) ...;
}
```

Why can I use == to compare `int`s, but it is it usually wrong to use it to compare strings?

Equivalence...

- Scheme has three equivalence predicates `eq?`, `eqv?` and `equal?`.
- `eq?` is the pickiest of the three, then comes `eqv?`, and last `equal?`.
- In other words,
  - If `(equal? a b)` returns `#t`, then so will `(eq? a b)` and `(eqv? a b)`.
  - If `(eqv? a b)` returns `#t`, then so will `(eq? a b)`..
- `(equal? a b)` generally returns `#t` if `a` and `b` are structurally the same, i.e. print the same.

Equivalence. . .

`(eqv? a b)` returns #t if:

- `a` and `b` are both `#t` or both `#f`.
- `a` and `b` are both symbols with the same name.
- `a` and `b` are both the same number.
- `a` and `b` are strings that denote the same locations in the store.

```
> (define S "hello")
> (eqv? S S)
true
> (eqv? "hello" "hello")
false
> (eqv? 'hello 'hello)
true
```

Equivalence...

- `(equal?  a b)` returns #t if `a` and `b` are strings that print the same.
- This is known as structural equivalence.

```
> (equal?  "hello" "hello")
true
> (equal?  alice bob)
false
> (define alice1 (
           make-person "alice" 'female '1979))
> (define alice2 (
           make-person "alice" 'female '1979))
> (equal?  alice1 alice2)
true
```

LIST PROCESSING

Constructing Lists

- The most important data structure in Scheme is the list.
- Lists are constructed using the function `cons`:

```
(cons first rest)
```

`cons` returns a list where the first element is `first`, followed by the elements from the list `rest`.

```
> (cons 'a '())
(a)
> (cons 'a (cons 'b '()))
(a b)
> (cons 'a (cons 'b (cons 'c '())))
(a b c)
```

Constructing Lists...

- There are a variety of short-hands for constructing lists.
- Lists are heterogeneous, they can contain elements of different types, including other lists.

```
> '(a b c)
(a b c)
> (list 'a 'b 'c)
(a b c)

> '(1 a "hello")
(1 a "hello")
```

Examining Lists

- `(car L)` returns the first element of a list. Some implementations also define this as `(first L)`.
- `(cdr L)` returns the list `L`, without the first element. Some implementations also define this as `(rest L)`.
- Note that `car` and `cdr` do not destroy the list, just return its parts.

```
> (car '(a b c))
'a
> (cdr '(a b c))
'(b c)
```

Examining Lists…

- Note that `(cdr L)` always returns a list.

```
> (car (cdr '(a b c)))
'b
> (cdr '(a b c))
'(b c)
> (cdr (cdr '(a b c)))
'(c)
> (cdr (cdr (cdr '(a b c))))
'()
> (cdr (cdr (cdr (cdr '(a b c)))))
error
```

Examining Lists...

- A shorthand has been developed for looking deep into a list:

  ```
  (clist of "a" and "d"r L)
  ```

  Each `"a"` stands for a `car`, each `"d"` for a `cdr`.
- For example, `(caddar L)` stands for

  ```
  (car (cdr (cdr (car L))))
  ```

```
> (cadr '(a b c))
'b
> (cddr '(a b c))
'(c)
> (caddr '(a b c))
'c
```

Lists of Lists

- Any S-expression is a valid list in Scheme.
- That is, lists can contain lists, which can contain lists, which...

```
> '(a (b c))
(a (b c))
> '(1 "hello" ("bye" 1/4 (apple)))
(1 "hello" ("bye" 1/4 (apple)))
> (caaddr '(1 "hello" ("bye" 1/4 (apple))))
"bye"
```

List Equivalence

- `(equal?  L1 L2)` does a structural comparison of two lists, returning `#t` if they “look the same”.
- `(eqv?  L1 L2)` does a “pointer comparison”, returning `#t` if two lists are “the same object”.

```
> (eqv?  '(a b c) '(a b c))
false
> (equal?  '(a b c) '(a b c))
true
```

List Equivalence…

- This is sometimes referred to as deep equivalence vs. shallow equivalence.

```
> (define myList '(a b c))
> (eqv?  myList myList)
true
> (eqv?  '(a (b c (d))) '(a (b c (d))))
false
> (equal?  '(a (b c (d))) '(a (b c (d))))
true
```

Predicates on Lists

- `(null? L)` returns `#t` for an empty list.
- `(list? L)` returns `#t` if the argument is a list.

```
> (null? '())
#t
> (null? '(a b c))
#f
> (list? '(a b c))
#t
> (list? "(a b c)")
#f
```

List Functions — Examples. . .

```
> (memq 'z '(x y z w))
#t
> (car (cdr (car '((a) b (c d)))))
(c d)
> (caddr '((a) b (c d)))
(c d)
> (cons 'a '())
(a)
> (cons 'd '(e))
(d e)
> (cons '(a b) '(c d))
((a b) (c d))
```

Recursion over Lists — cdr-recursion

- Compute the length of a list.
- This is called cdr-recursion.

```
(define (length x)
    (cond
        [(null? x) 0]
        [else (+ 1 (length (cdr x)))]
    )
)

> (length '(1 2 3))
3
> (length '(a (b c) (d e f)))
3
```

Recursion over Lists — car-cdr-recursion

- Count the number of atoms in an S-expression.
- This is called car-cdr-recursion.

```
(define (atomcount x)
   (cond
      [(null?  x) 0]
      [(list?  x)
            (+ (atomcount (car x))
               (atomcount (cdr x)))]
      [else 1]
   ))
> (atomcount '(1))
1
> (atomcount '("hello" a b (c 1 (d))))
6
```

Recursion Over Lists — Returning a List

- Map a list of numbers to a new list of their absolute values.
- In the previous examples we returned an atom — here we're mapping a list to a new list.

```
(define (abs-list L)
   (cond
      [(null?  L) '()]
      [else (cons (abs (car L))
                  (abs-list (cdr L)))]
   )
)

> (abs-list '(1 -1 2 -3 5))
(1 1 2 3 5)
```

Recursion Over Two Lists

- `(atom-list-eq? L1 L2)` returns `#t` if `L1` and `L2` are the same list of atoms.

```
(define (atom-list-eq? L1 L2)
   (cond
      [(and (null? L1) (null? L2)) #t]
      [(or (null? L1) (null? L2)) #f]
      [else (and
         (atom? (car L1))
         (atom? (car L2))
         (eqv? (car L1) (car L2))
         (atom-list-eq? (cdr L1) (cdr L2)))]
   )
)
```

Recursion Over Two Lists…

```
> (atom-list-eq?  '(1 2 3) '(1 2 3))
#t
> (atom-list-eq?  '(1 2 3) '(1 2 a))
#f
```

Append

```
(define (append L1 L2)
    (cond
        [(null?  L1) L2]
        [else
            (cons (car L1)
                (append (cdr L1) L2))]
    )
)

> (append '(1 2) '(3 4))
(1 2 3 4)
> (append '() '(3 4))
(3 4)
> (append '(1 2) '())
(1 2)
```

Deep Recursion — `equal?`

```
(define (equal? x y)
    (or (and (atom? x) (atom? y) (eq? x y))
        (and (not (atom? x))
             (not (atom? y))
             (equal? (car x) (car y))
             (equal? (cdr x) (cdr y)))))
```

```
> (equal? 'a 'a)
#t
> (equal? '(a) '(a))
#t
> (equal? '((a)) '((a)))
#t
```

Patterns of Recursion — cdr-recursion

- We process the elements of the list one at a time.
- Nested lists are not descended into.

```
(define (fun L)
   (cond
      [(null?  L) return-value]
      [else ...(car L) ...(fun (cdr L)) ...]
   )
)
```

Patterns of Recursion — car-cdr-recursio

- We descend into nested lists, processing every atom.

```
(define (fun x)
   (cond
      [(null?  x) return-value]
      [(atom?  x) return-value]
      [(list?  x)
            ...(fun (car x)) ...
            ...(fun (cdr x)) ...]
      [else return-value]
   ))
```

Patterns of Recursion — Maps

- Here we map one list to another.

```
(define (map L)
   (cond
      [(null?  L) '()]
      [else (cons (...(car L) ...)
                  (map (cdr L)))]
   )
)
```

Example: Binary Trees

- A binary tree can be represented as nested lists:

```
(4 (2 () () ( 6 ( 5 () ()) ()))) 
```

- Each node is represented by a triple

  `(`*`data left-subtree right-subtree`*`)`

- Empty subtrees are represented by `()`.

Example: Binary Trees. . .

```
(define (key tree) (car tree))
(define (left tree) (cadr tree))
(define (right tree) (caddr tree))

(define (print-spaces N)
   (cond
      [(= N 0) ""]
      [else (begin
         (display " ")
         (print-spaces (- N 1)))]))

(define (print-tree tree)
   (print-tree-rec tree 0))
```

Example: Binary Trees...

```
(define (print-tree-rec tree D)
   (cond
      [(null?  tree)]
      [else (begin
         (print-spaces D)
         (display (key tree)) (newline)
         (print-tree-rec (left tree) (+ D 1))
         (print-tree-rec (right tree) (+ D 1))
)]))
```

```
> (print-tree '(4 (2 () ()) (6 (5 () ()) ())))
4
   2
   6
      5
```

Binary Trees using Structures

- We can use structures to define tree nodes.

```
(define-struct node (data left right))

(define (tree-member x T)
    (cond
        [(null?  T) #f]
        [(= x (node-data T)) #t]
        [(< x (node-data T))
            (tree-member x (node-left T))]
        [else
            (tree-member x (node-right T))]
    )
)
```

Binary Trees using Structures…

```
(define tree
    (make-node 4
        (make-node 2 ’() ’())
        (make-node 6
            (make-node 5 ’() ’())
            (make-node 9 ’() ’())))))

> (tree-member 4 tree)
true
> (tree-member 5 tree)
true
> (tree-member 19 tree)
false
```

Homework

- Write a function `swapFirstTwo` which swaps the first two elements of a list. Example: `(1 2 3 4)` $\Rightarrow$ `(2 1 3 4)`.
- Write a function `swapTwoInLists` which, given a list of lists, forms a new list of all elements in all lists, with first two of each swapped. Example: `((1 2 3) (4) (5 6))` $\Rightarrow$ `(2 1 3 4 6 5)`.

HIGHER-ORDER FUNCTIONS

Higher-Order Functions

- A function is higher-order if
  1. it takes another function as an argument, or
  2. it returns a function as its result.
- Functional programs make extensive use of higher-order functions to make programs smaller and more elegant.
- We use higher-order functions to encapsulate common patterns of computation.

Higher-Order Functions: `map`

- Map a list of numbers to a new list of their absolute values.
- Here's the definition of `abs-list` from a previous lecture:

```
(define (abs-list L)
    (cond
        [(null?  L) '()]
        [else (cons (abs (car L))
                        (abs-list (cdr L)))]
    )
)

> (abs-list '(1 -1 2 -3 5))
(1 1 2 3 5)
```

Higher-Order Functions: `map`...

- This type of computation is very common.
- Scheme therefore has a built-in function

  `(map f L)`

  which constructs a new list by applying the function f to every element of the list L.

$$\texttt{(map f '(}e_1\ e_2\ e_3\ e_4\texttt{))}$$
$$\Downarrow$$
$$\texttt{((f }e_1\texttt{) (f }e_2\texttt{) (f }e_3\texttt{) (f }e_4\texttt{))}$$

Higher-Order Functions: `map`...

- `map` is a higher-order function, i.e. it takes another function as an argument.

```
(define (addone a) (+ 1 a))

> (map addone '(1 2 3))
(2 3 4)

> (map abs '(-1 2 -3))
(1 2 3)
```

Higher-Order Functions: `map`...

- We can easily define `map` ourselves:

```
(define (mymap f L)
   (cond
      [(null?  L) '()]
      [else
         (cons (f (car L)) (mymap f (cdr L)))]))
```

```
> (mymap abs '(-1 2 -3))
(1 2 3)
```

Higher-Order Functions: `map`...

- If the function takes n arguments, we give `map` n lists of arguments:

```
> (map string-append
           '("A" "B" "C") '("1" "2" "3"))
("A1" "B2" "C3")

> (map + '(1 2 3) '(1 2 3))
(list 2 4 6)

> (map cons '(a b c) '((1) (2) (3)))
((a 1) (b 2) (c 3))
```

Lambda Expressions

- A lambda-expression evaluates to a function:

  ```
  (lambda (x) (* x x))
  ```

  `x` is the function's formal parameter.
- Lambda-expressions don't give the function a name — they're anonymous functions.
- Evaluating the function:

  ```
  > ((lambda (x) (* x x)) 3)
  9
  ```

Higher-Order Functions: `map`...

- We can use `lambda`-expressions to construct anonymous functions to pass to `map`. This saves us from having to define auxiliary functions:

```
(define (addone a) (+ 1 a))

> (map addone '(1 2 3))
(2 3 4)

> (map (lambda (a) (+ 1 a)) '(1 2 3))
(2 3 4)
```

Higher-Order Functions: `filter`

- The filter-function applies a predicate (boolean-valued function) `p` to all the elements of a list.
- A new list is returned consisting of those elements for which `p` returns `#t`.

```
(define (filter p L)
   (cond
      [(null?  L) '()]
      [(p (car L))
            (cons (car L) (filter p (cdr L)))]
      [else (filter p (cdr L))]))

> (filter (lambda (x) (> x 0)) '(1 -2 3 -4))
(1 3)
```

Higher-Order Functions: `fold`

Consider the following two functions:

```
(define (sum L)
   (cond
      [(null?  L) 0]
      [else (+ (car L) (sum (cdr L)))]))
(define (concat L)
   (cond
      [(null?  L) "" ]
      [else (string-append (car L) (concat (cdr L)))]))

> (sum '(1 2 3))
6
> (concat '("1" "2" "3"))
"123"
```

Higher-Order Functions: `fold`...

- The two functions only differ in what operations they apply (+ vs. string-append, and in the value returned for the base case (0 vs. "").
- The `fold` function abstracts this computation:

```
(define (fold L f n)
   (cond
      [(null? L) n]
      [else (f (car L) (fold (cdr L) f n))]))

> (fold '(1 2 3) + 0)
6
> (fold '("A" "B" "C") string-append "")
"ABC"
```

Higher-Order Functions: `fold`

- In other words, `fold` folds a list together by successively applying the function `f` to the elements of the list `L`.

  (apply f '(e_1 e_2 e_3 e_4)) $\Rightarrow$

  (f e_1 (f e_2 (f e_3 e_4)))

METACIRCULAR INTERPRETATION

Introduction

- In this lecture I’m going to show how you can define Scheme by writing a metacircular interpreter for the language, i.e. an interpreter for Scheme written in Scheme.
- Before we can do that, we first need to learn a few more this about the language

Let Expressions

- A let-expression binds names to values:

  ```
  (let ((name₁ value₁) (name₂ value₂) ...)
        expression)
  ```

- The first argument to `let` is a list of `(name value)` pairs. The second argument is the expression to evaluate.

```
> (let ((a 3) (b 4) (square (lambda (x)(* x x)))
           (plus +))
        (sqrt (plus (square a) (square b))))
5.0
```

Let Expressions. . .

- `Let`-expressions can be nested:

```
> (let ((x 5) (c 4))
    (let ((v (* 4 x))
          (t (* 2 c)))
         (+ v t)))
28
```

Imperative Features

- Scheme is an impure functional language.
- I.e., Scheme has imperative features.
- I.e., in Scheme it is possible to program with side-effects.

`(set! var value)` Change the value of `var` to `value`.

`(set-car! var value)` Change the `car`-field of the cons-cell `var` to `value`.

`(set-cdr! var value)` Change the `cdr`-field of the cons-cell `var` to `value`.

Imperative Features. . .

- Example:

```
> (let ((x 2) (l '(a b)))
    (set!  x 3)
    (set-car!  l '(c d))
    (set-cdr!  l '(e))
    (display x) (newline)
    (display l) (newline))
3
((c d) e)
```

Dotted Pairs

- S-expressions are constructed using dotted pairs.
- It is implemented as a `struct` (called a cons-cell) consisting of two fields (the size of a machine word) called `car` and `cdr`.
- We can manipulate these fields directly:

```
> '(1 . 2)
(1 . 2)
> (cons "stacy's" "mom")
("stacy's" . "mom")
> '(1 . (2 . 3))
(1 2 . 3)
> (cons 1 2)
(1 . 2)
```

Dotted Pairs. . .

- When the second part of a dottend pair (the `cdr`-field) is a list, and the innermost `cdr`-field is the empty list, we get a “normal” Scheme list:

```
> '(1 . ())
(1)
> '(1 . (2 . ()))
(1 2)
> '(1 . (2 3))
(1 2 3)
```

Dotted Pairs...

- We can use `set-car!` and `set-cdr!` to manipulate the fields of a `cons`-cell directly:

```
> (define x '(1 . 2))
> (set-car! x 'a)
> x
(a . 2)
> (set-cdr! x '(2 3))
> x
(a 2 3)
```

Dotted Pairs. . .

- `(cons A B)` can be thought of as first creating a `cons`-cell on the heap (using `malloc`, for example), and then setting the `car` and `cdr` fields to `A` and `B`, respectively:

```
> (define x (cons 0 0))
> x
(0 . 0)
> (set-car! x '1)
> (set-cdr! x '())
> x
(1)
```

Loops

Scheme's "for-loop" `do` takes these arguments:

1. A list of triples (*var init update*) which declares a variable *var*, with an initial value *init*, and which gets updated using the expression *update*, on each iteration;
2. A pair (*termination_cond return_value*) which gives the termination condition and return value of the loop; and
3. a loop body:

```
(do ((var1 init1 update1)
     (var12 init2 update2)
           ...
     )
     (termination_cond return_value)
   loop_body
)
```

Loops…

- Sum the numbers 1 to 4, printing out intermediate results:

```
> (do ((i 1 (+ i 1))
    (sum 0 (+ sum i)))
    ((= i 5) sum)
    (display sum)
    (newline)
)
0
1
3
6

10
```

Association Lists

- Association lists are simply lists of *key*-*value* pairs that can be searched sequentially:

```
> (assoc 'bob '((bob 22) (joe 32) (bob 3)))
(bob 22)
```

- The list is searchedy the list from beginning to end, returning the first pair with a matching key:

  `(assoc key alist)` Search for *key*; compare using `equal?`.

  `(assq key alist)` Search for *key*; compare using `eq?`.

  `(assv key alist)` Search for *key*; compare using `eqv?`.

Association Lists. . .

```
> (define e '((a 1) (b 2) (c 3)))
> (assq 'a e)
(a 1)
> (assq 'b e)
(b 2)
> (assq 'd e)
#f
> (assq (list 'a) '(((a)) ((b)) ((c))))
#f
> (assoc (list 'a) '(((a)) ((b)) ((c))))
((a))
> (assv 5 '((2 3) (5 7) (11 13)))
(5 7)
```

Association Lists...

- We can actually have more than one value:

```
> (assoc 'bob '((bob 5 male)
               (jane 32 'female)))
(bob 5 male)
```

Apply

- Apply returns the result of applying its first argument to its second argument.

```
> (apply + '(6 7))
13
> (apply max '(2 5 1 7))
7
```

Eval

- (eval *arg*) evaluates its argument.

```
> (eval '(+ 4 5))
9
> (eval '(cons 'a '(b c))) (a b c)
```

Eval. . .

- `eval` and `quote` are each other's inverses:

```
> (eval ''(+ 4 5))
(+ 4 5)
> (eval (eval ''(+ 4 5)))
9
> (eval (eval (eval '''(+ 4 5))))
9
```

Programs as Data

- Scheme is homoiconic, self-representing, i.e. programs and data are both represented the same (as S-expressions).
- This allows us to write programs that generate programs - useful in AI, for example.

```
> (define x 'car)
> (define y ''(a b c))
> (define p (list x y))
> p
(car '(a b c))
> (eval p)
a
```

Evaluation Order

- So far, we have said that to evaluate an expression `(op arg1 arg2 arg3)` we first evaluate the arguments, then apply the operator `op` to the resulting values.
- This is known as applicative-order evaluation.
- Example:

```
(define (double x) (* x x))

> (double (* 3 4))
   ⇒ (double 12)
   ⇒ (+ 12 12)
   ⇒ 24
```

Evaluation Order. . .

- This is not the only possible order of evaluation
- In normal-order evaluation parameters to a function are always passed unevaluated.
- This sometimes leads to extra work:

```
(define (double x) (* x x))

> (double (* 3 4))
   ⇒ (+ (* 3 4) (* 3 4)))
   ⇒ (+ 12 (* 3 4))
   ⇒ (+ 12 12)
   ⇒ 24
```

Evaluation Order. . .

- Applicative-order can sometimes also lead to more work than normal-order:

```
(define (switch x a b c)
   (cond
      ((< x 0) a)
      ((= x 0) b)
      ((> x 0) c)))

> (switch -1 (+ 1 2) (+ 2 3) (+ 3 4))
```

- Here, applicative-order evaluates all the arguments, although only one value will ever be needed.

Evaluation Order...

- Ordinary Scheme functions (such as `+`, `car`, etc) use applicative-order evaluation.
- Some special forms (`cond`, `if`, etc) must use normal order since they need to consume their arguments unevaluated:

```
> (if #t (display 5) (display 6))
5
> (cond (#f (display 5))
        (#f (display 6))
        (#t (display 7)))
7
```

A Metacircular Interpreter

- One way to define the semantics of a language (the effects that programs written in the language will have), is to write a metacircular interpreter.
- I.e, we define the language by writing an interpreter for it, in the language itself.
- A metacircular interpreter for Scheme consists of two mutually recursive functions, `mEval` and `mApply`:

```
(define (mEval Expr)
    ...
)
(define (mApply Op Args)
    ...
)
```

A Metacircular Interpreter. . .

- We want to be able to call our interpreter like this:

```
> (mEval (+ 1 2))
3
> (mEval (+ 1 (* 3 4)))
13
> (mEval (quote (2 3)))
(2 3)
> (mEval (car (quote (1 2))))
1
```

A Metacircular Interpreter…

```
> (mEval (cdr (quote (1 2))))
(2)
> (mEval (cons (quote 5) (quote (1 2))))
(5 1 2)
> (mEval (null?  (quote (1 2))))
#f
> (mEval (null?  (quote ())))
#t
> (mEval (if (eq?  1 1) 5 6))
5
```

A Metacircular Interpreter…

- `mEval` handles primitive special forms (`lambda`, `if`, `const`, `define`, `quote`, etc), itself.
- Note that, for these forms, we must use normal-order evaluation.
- For other expressions, `mEval` evaluates all arguments and calls `mApply` to perform the required operation:

A Metacircular Interpreter…

```
(define (mEval Expr)
   (cond
      [(null?  Expr) '()]
      [(number?  Expr) Expr]
      [(eq?  (car Expr) 'if)
         (mEvalIf (cadr Expr)
                  (caddr Expr)
                  (cadddr Expr))]
      [(eq?  (car Expr) 'quote) (cadr Expr)]
      [else (mApply (car Expr)
                  (mEvalList (cdr Expr)))]
   )
)
```

A Metacircular Interpreter. . .

- `mApply` checks if the operation is one of the builtin primitive ones, and if so performs the required operation:

```
(define (mApply Op Args)
   (case Op
      [(car) (caar Args)]
      [(cdr) (cdar Args)]
      [(cons) (cons (car Args) (cadr Args))]
      [(eq?)  (eq?  (car Args) (cadr Args))]
      [(null?)  (null?  (car Args))]
      [(+) (+ (car Args) (cadr Args))]
      [(*) (* (car Args) (cadr Args))]
   )
)
```

A Metacircular Interpreter…

- Some auxiliary functions:

```
(define (mEvalIf b t e)
    (if (mEval b) (mEval t) (mEval e))
)

(define (mEvalList List)
    (cond
        [(null? List) '()]
        [else (cons (mEval (car List))
                (mEvalList (cdr List)))]
    )
)
```

A Metacircular Interpreter…

- Note that this little interpreter lacks many of Scheme’s functions.
- We don’t have symbols, `lambda`, `define`.
- We can’t define or invoke user-defined functions.
- There are no way to define or lookup variables, local or global. To do that, `mEval` and `mApply` pass around environments (*association lists*) of variable/value pairs.